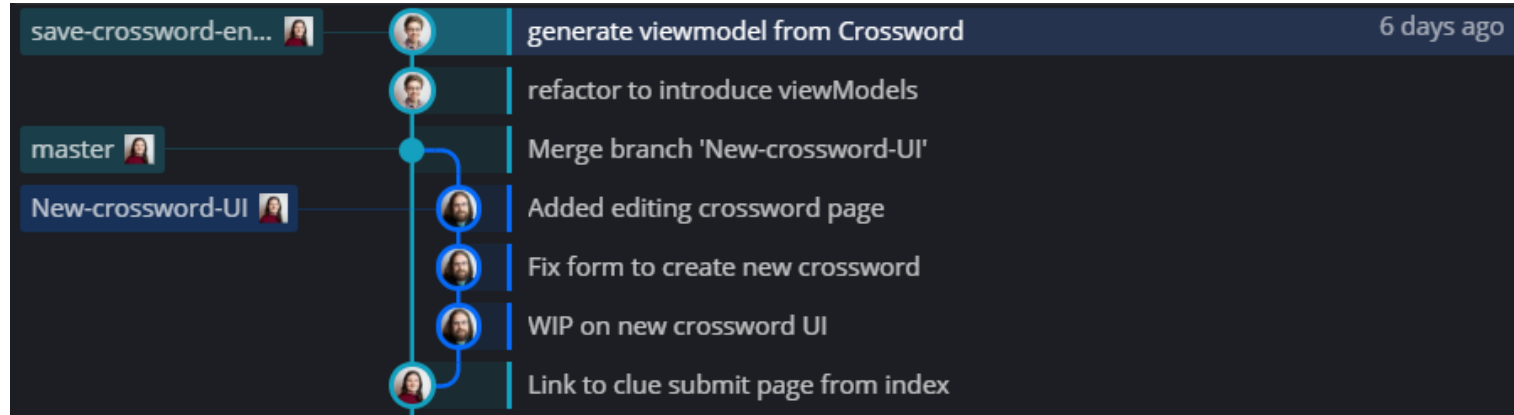


# Basic Git - Mentimeter

## Part 2: Version Control In Git

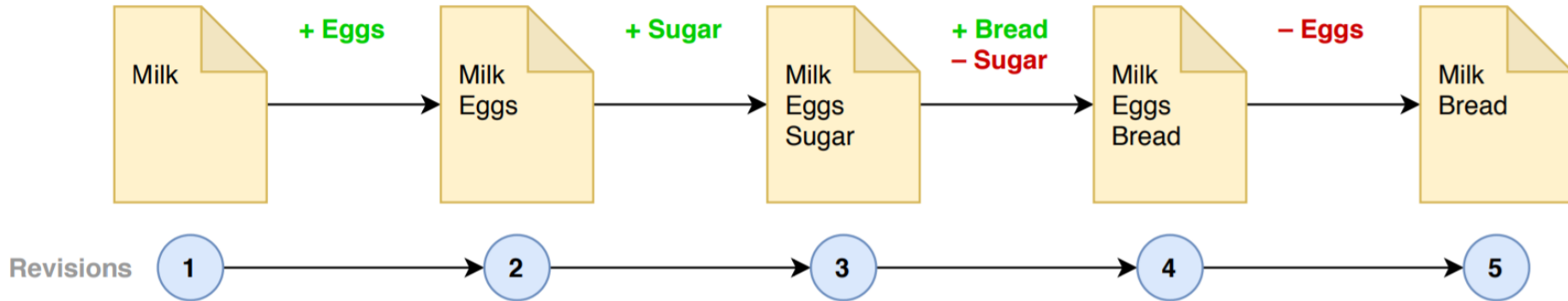
### Purpose of Git

Git is a version control system that allows you to track changes to your codebase over time, examine the differences between the current state of your codebase and the last known working version, as well as check out your codebase as it existed at various points in time:



# Commits

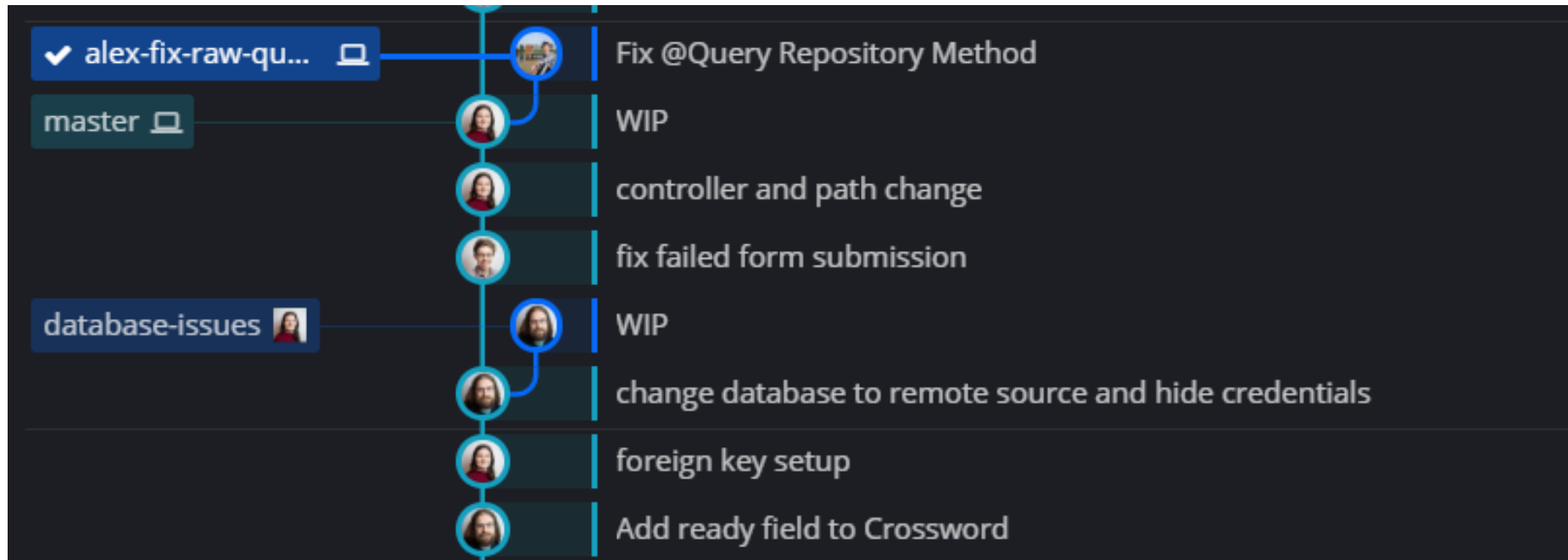
"Commits" are snapshots of your code that you've saved (like working on an important document and having copies called "My doc V1.docx", "My doc V2.docx", "My doc V3.docx", etc).



Every time you make a commit, you build on the previous one, making a chain of commits in a sort of timeline, retaining the ability to "time travel" to any point in time.

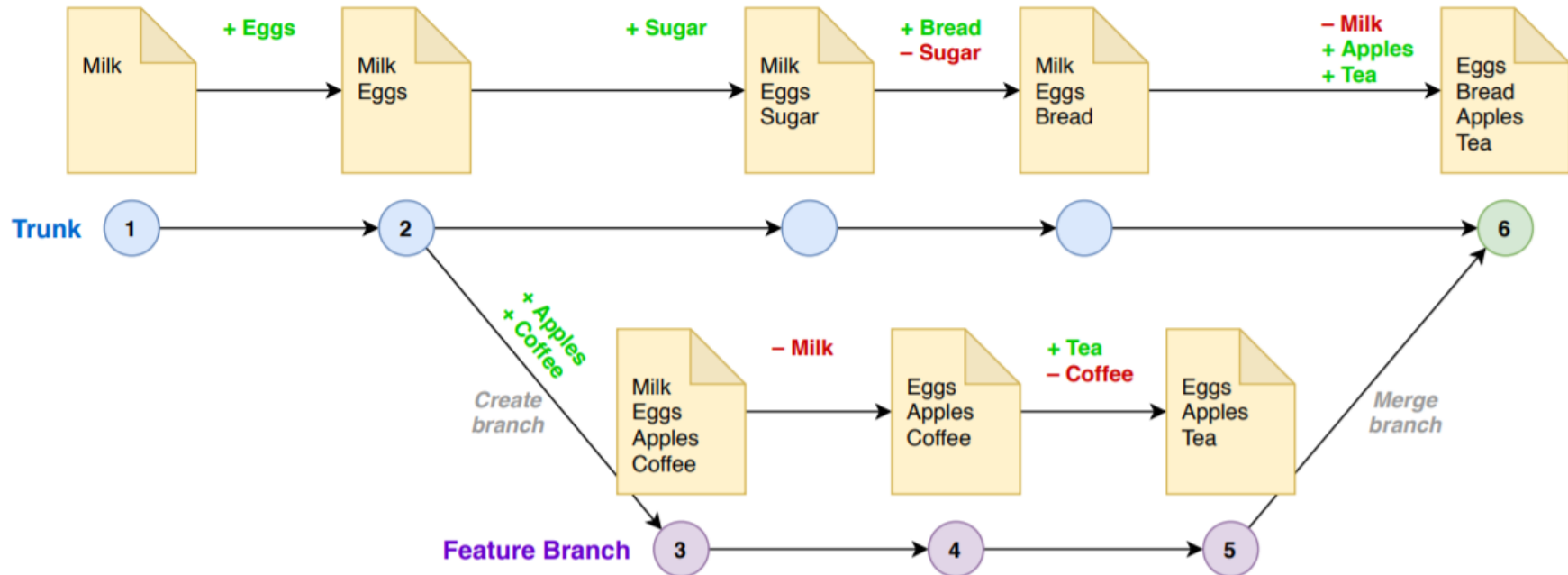
# Branches

You can also make "Branches" - parallel universes in which your codebase looks slightly different. You can travel to different branches too.



# Merging Of Branches

Branches can be merged into each other (typically creating a “merge commit”)



# Basic Git Commands

Here is a summary of all the git commands you need to know about right now:

```
$ git status
$ git init
$ git add <path to file>
$ git commit -m "<commit message>"
$ git branch <new branch name>
$ git checkout <target branch>
$ git remote add <remote name> <remote url>
$ git fetch
$ git pull
$ git push
$ git merge <branch to be merged into current branch>
```

## Part 2: Exercise

### The Goal

Setting up a GitHub repository with your code from part 1.

You will:

- Create a local repository, with some initial commits and branches
- Create an online repository for your code and push up code from your local repository
- (Extension) Try resetting & reverting
- (Extension) Create some pull requests for your repository
- (Extension) Create (!) and resolve a merge conflict

# Step 1

From the directory of your code from part 1, initialise a git repository and commit everything you have as an initial commit.

- If you don't have git on your machine at this point please download Git from <https://git-scm.com/downloads>
  - a) Add a readme file (called README.md) that describes how to use your code (in simple plaintext) and make a new commit with a message "Added README"
  - b) Add a text file with the name NewFile.txt to the directory of your local repository and create an appropriate commit.

You can ask a trainer to review the state of your git repository if you'd like to confirm before proceeding to step 2.

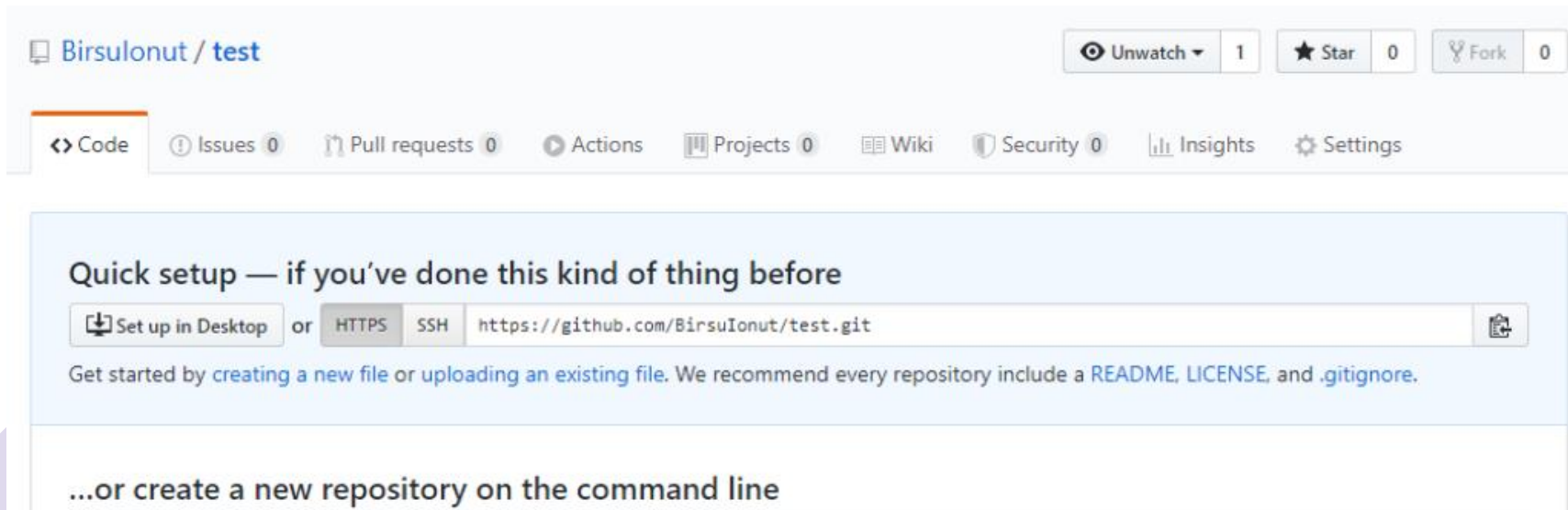
*Tip: You may want to run a git visualiser tool like gitk or GitKraken to track your commits/branches*



## Step 2 (Page 1 of 2)

Next, let's push up the code from parts 1 & 2 to a remote repository:

- a) If you don't have an account already, create one on <https://github.com/>
  - You may want to create a personal one if you only have a work account
- b) Create a new repository, giving it a name and leaving everything else untouched. You should now see this page:



The screenshot shows the GitHub interface for a new repository named 'test' by user 'Birsulonut'. At the top, there are buttons for 'Unwatch' (1), 'Star' (0), and 'Fork' (0). Below this is a navigation bar with links for 'Code', 'Issues' (0), 'Pull requests' (0), 'Actions', 'Projects' (0), 'Wiki', 'Security' (0), 'Insights', and 'Settings'. The main content area has a heading 'Quick setup — if you've done this kind of thing before'. Below the heading are three tabs: 'Set up in Desktop', 'HTTPS', and 'SSH'. The 'SSH' tab is selected, showing the URL 'https://github.com/BirsuIonut/test.git'. Below the tabs, there is a text prompt: 'Get started by creating a new file or uploading an existing file. We recommend every repository include a README, LICENSE, and .gitignore.' At the bottom, there is a section titled '...or create a new repository on the command line'.

## Step 2 (Page 2 of 2)

- d) Follow the instructions under "**...or push an existing repository from the command line**"
- e) You will be prompted to provide your GitHub username and password in the terminal
  - If you prefer not to enter your credentials over a terminal you can create a SSH key to use with your GitHub account. This will involve running:

```
$ ssh-keygen -t rsa -b 4096 -C "your_email@example.com"
```

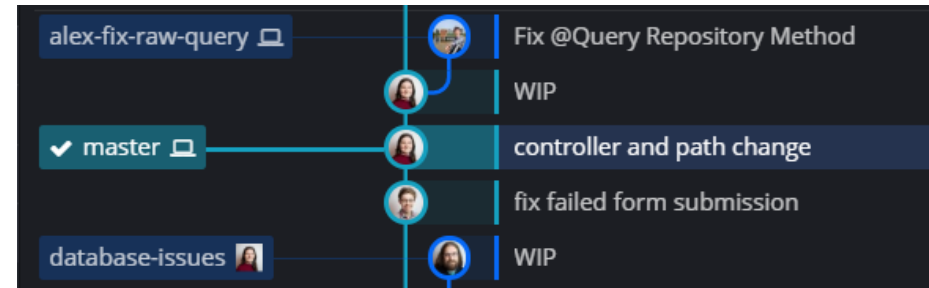
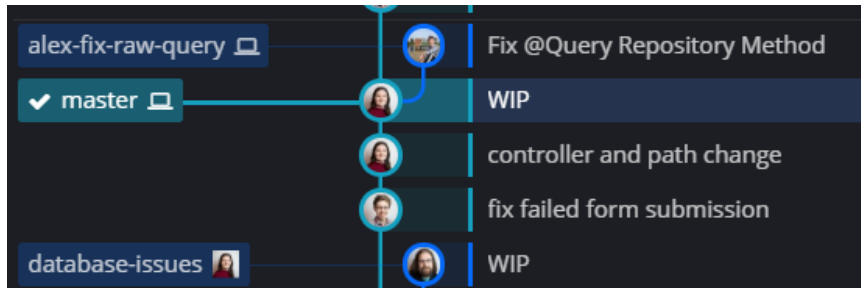
And uploading the key to GitHub as described [here](#)  
(Ask a tutor if you want more details)

Refresh the page and now you should be able to see your pushed code. Please show the results to your tutor when we come by.

## Resetting

In Git, branches are merely a movable pointer/label on a commit. The label moves when you commit to that branch.

Resetting is a branch operation that moves the label directly:



## Resetting (Continued)

The command for resetting is

```
$ git reset [--soft] [--hard] <location (can be branch, commit hash, etc.)>
```

A **soft** reset (--soft) will only move the position of branch label (meaning that git will now notice a lot of staged changes)

Using a **hard** reset (--hard) will change all tracked files to match the new commit along with removing any new files that were added since.

The location is very flexible and you can even specify commits relative to the position of the current branch:

```
$ git reset --hard HEAD~2 # Move the current branch back 2 commits
```

## Step 3

### Reset

Create a new branch called "reset", checkout this branch and then reset the branch to an earlier commit (try a soft and then a hard reset).

- Can you see the difference between a soft and hard reset?
- You can reset using a relative commit (e.g. HEAD~1) or by finding out the commit id for an existing commit.
  - Try finding the id for an existing commit – your visualizer will offer one way, or try using `git log`. (Hint: “q” will quit the log!)

### Revert

Reverting a commit undoes the changes made in that commit.

Checkout the master branch. Next create and checkout a branch called "revert". Finally create a revert commit for the file addition (hint: use `git revert`)

Look at your git graph in a visualizer – has it done what you'd expect?

## Step 4

Next, let's create a pull request:

- a) Create a new branch (you can call it something like **pow-operator**)
- b) Change the code from part 1 by adding a new operator, **pow**

```
calc ^ 2 5
```

- c) Afterwards, make a commit and push the new branch to origin
- d) On the repository page, go to branches and create a **New pull request** (you can also add a comment about the changes you made). For now, **do not merge it**.

## Step 5 (Page 1 of 2)

Finally, let's resolve a merge conflict:

- a) Checkout master and create a new branch, this time called **mod-operator**.
- b) Change the code from part 1 by adding a new operator, **mod**.

```
calc % 2 5
```

For the purposes of this exercise, you should have some overlapping work with the **pow-operator** branch (e.g. new work inside the same if-clauses, or adding functions at the end of the files, etc).

- c) Make a commit and push the new branch to origin.

## Step 5 (Page 2 of 2)

- d) Merge the **pow-operator** branch (you can also safely delete the branch, as github will suggest you).
- e) Create a new pull request, this time for **mod-operator** branch. This time though, you will get some warnings about some existing conflicts.
- f) Go ahead and press **Resolve conflicts** button and resolve the conflicts GitHub couldn't solve on his own. For each file, when you finished, press **Mark as resolved**.
- g) After resolving all the conflicts, commit merge and merge the branch into master.



# Rebasing Branches

**Rebasing** is where a branch's commits are replayed on top of another branch. It can be used as an alternative to merging branches.

This can be useful for simplifying a repository's commit history but does change that history permanently (so is not without risk).

The most common use for this is where you want to replay a feature branch on top of the latest state of the master branch:



## Rebasing Branches (Continued)

The command for rebasing is:

```
$ git rebase <target branch> # Assumes checked-out branch is the branch to be rebased
```

```
$ git rebase <target branch> <branch to rebase> # Shorthand to checkout the branch first
```

**Interactive rebases** provide customisation of how the branch's commits should be replayed (typically an editor will open providing you with instructions on your customisation options):

```
$ git rebase -i <any rebase details>
```

Warning: Like merging, rebasing can lead to merge conflicts!

## Step 6

Finally, let's try an interactive rebase:

- a) Create a branch called pre-merge that 2 commits behind master (i.e. before the 2 merge commits that were created in step 4).
- b) Create a new branch from master called ***squashed-merge-commits***.
- c) Interactively rebase ***squashed-merge-commits*** onto pre-merge, combining the 2 merge commits into 1 and changing the commit message to reflect both commits.

# Part 3: Python libraries and the Flask web framework

## Recap

Python has a rich ecosystem of libraries and frameworks that have been written to solve common requirements.

Packages can be installed via the pip package manager:

```
$ pip install some-package-name
```

And can then be imported as modules in python scripts:

```
1 from flask import Flask
2
3 app = Flask(__name__)
```

Python also has built in modules that don't need to be installed via pip (e.g. datetime).

# Web Applications

Refers to any computer program that contains dynamic logic and deals with user-generated data, which users interact with via the World Wide Web (either through their browser or other client software). This could be

- A website with lots of dynamic content
- A desktop app that interfaces with the web (e.g. most messaging apps)

Web apps typically rely upon a client-server architecture where the logic and content is provided by an application running on a server, but is displayed and runs in a user's web browser (the client).

# Flask

Flask is a popular web application framework that provides features such as:

- Running a local server

```
1  from flask import Flask
2
3  app = Flask(__name__)
4
5  @app.route('/hello')
6  def hello_world():
7      return 'Hello World!'
```

```
$ flask run
```

## Flask (continued)

- HTTP Request routing

```
1 @app.route('/books')
2 def get_books():
3     # Code to fetch all book entries from the database and return their details.
4
5 @app.route('/books/<id>')
6 def get_book(id):
7     # Code to fetch the book entry with matching id from the database and return its details.
8
9 @app.route('/books', methods=['POST'])
10 def add_book():
11     # Code to create a new book entry in the database.
```

## Part 3: Exercise

### The Goal

Installing the Flask library and setting up some endpoints covering the use of:

- Serving HTML pages
- Templating
- Submitting and processing forms

Installing a command line library argparse and setting up a command line interface (CLI) to generate data



## Step 1:

Install the [Flask](#) python library, as you would any package

```
$ pip install some-package-name
```

Check that you can run a simple Flask app like that on the right, using:

```
$ flask run
```

### Hints

- Name your Python file “app.py”
- When ready, visit <http://localhost:5000/hello> in a browser

```
1 from flask import Flask
2
3 app = Flask(__name__)
4
5 @app.route('/hello')
6 def hello_world():
7     return 'Hello World!'
```

## Step 2:

Next set up a route that serves the HTML page on the right.

You should serve this under the route `"/films/list"` on your flask web server.

### *Hints:*

- *Place the HTML file in a folder called `templates`*
- *Use [render template](#) from the flask library*
  - *See the example on the next slide if you're not sure how!*

```
<!doctype html>
<html>
  <head>
    <title>All films</title>
  </head>
  <body>
    <ul>
      <li>Flubber</li>
      <li>Jumanji</li>
      <li>Aladdin</li>
    </ul>
  </body>
</html>
```

# Flask Templating

```
books = [  
    { 'id': 1, 'title': 'Clean Code', 'authors': 'Robert C. Martin' },  
    { 'id': 2, 'title': 'The DevOps Handbook', 'authors': 'Gene Kim, Jez Humble, Patrick Debois, John Willis' },  
    { 'id': 3, 'title': 'The Phoenix Project', 'authors': 'Gene Kim, Kevin Behr, George Spafford' }  
]
```

```
1 @app.route('/books')  
2 def get_books():  
3     books = get_all_books_from_db() # Some function that returns the list of book entries from the database.  
4     return render_template('book_list.html', books=books)
```

This specifies the  
name that will be  
available within the  
HTML

And this refers to the  
variable that exists  
locally

# Flask Templating (continued)

## book\_list.html

```
1  <!doctype html>
2  <html>
3    <head>
4      <title>All Books</title>
5    </head>
6    <body>
7      <ul>
8        {% for book in books %}
9          <li>{{ book.title }} - {{ book.authors }}</li>
10         {% endfor %}
11      </ul>
12    </body>
13  </html>
```

If you're new to HTML, remember:

Inside the `<body>` is where we put elements we want to display on the page

`<ul>` is an “unordered list”

`<li>` is a “list item” – inside an unordered list, these will appear as bullet points

## Step 3:

Next, create a text file containing a list of films and ratings in the following format (known as “csv”):

```
Flubber,3  
Jumanji,5  
Aladdin,4
```

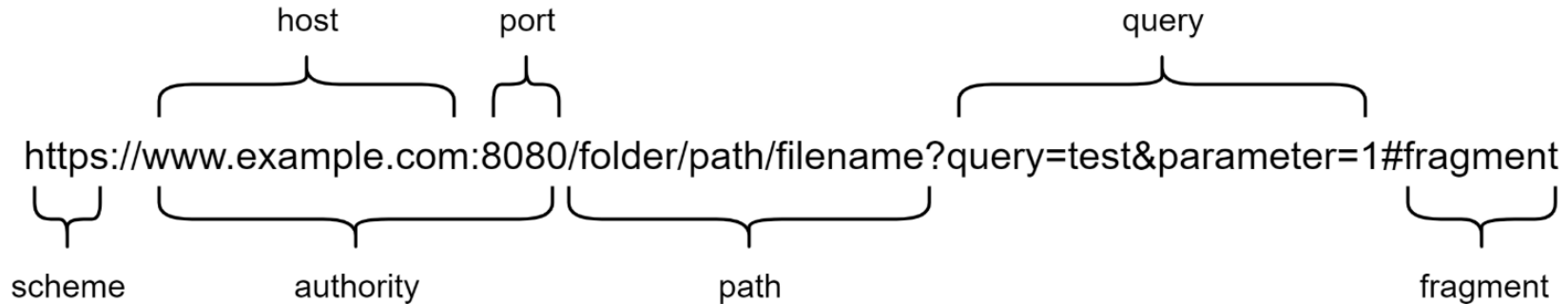
Serve a templated table of films along with their star ratings on the route "films/table”:

```
Film Stars  
Flubber 3  
Jumanji 5  
Aladdin 4
```

*Hint: We’ve covered how to do this in earlier slides!*

## Handling Query Parameters

Recall that query parameters are part of the anatomy of a URL:



They are often used as parameters to HTTP GET requests

# Reading Query Parameters In Flask

```
from flask import Flask, request

app = Flask(__name__)

@app.route("/show-message")
def echo():
    name = request.values.get("name", "")
    message = request.values.get("message", "")

    return f"Hey there {name}! You said {message}"
```

If you queried the path

```
/show-message?name=Alex&message=Hello%20Everyone!
```

The response would be

```
Hey there Alex! You said Hello Everyone!
```

## Step 4:

Now add support for filtering this list by star rating, e.g. "films/table?stars=3" should return just:

**Film Stars**  
Flubber 3

You can show the tutor your progress to confirm that everything is working as expected.



## Step 5:

Use the built in [argparse](#) python library and use it generate a command line app for submitting film reviews. Running this command should add “flubber,3” as a new line in your csv file.

```
python cli_app.py --film-name=flubber --stars=3
```

The app should support submitting multiple reviews (by repeatedly running the app) to build up a list of many films.

*Note: If you're having trouble getting argparse to work, feel free to use python's inbuilt “input” function instead to read values from the command line. There's also a tutorial for using the library [here](#).*

# Template Includes and Extends

The templating library for Flask, Jinja, has various mechanisms for composing HTML templates together.

The simplest is a direct inclusion of one template within another. For example you could have:

```
{% include 'header.html' %}  
    Body  
{% include 'footer.html' %}
```

Which includes 2 external templates corresponding to the head and footer respectively.

Please note that included templates can also use the same templated parameters as the master template.

## Template Includes and Extends (continued)

It is also possible for one template to “inherit” from another template using the extends keyword:

### Base Template

```
<!DOCTYPE html>
<html lang="en">
<head>
  <link rel="stylesheet" href="style.css" />
  <title>{% block title %}Default Title{% endblock %} - My Webpage</title>
</head>
<body>
  <div id="content">{% block content %}{% endblock %}</div>
</body>
</html>
```

### Child Template

```
{% extends "base.html" %}
{% block title %}Index{% endblock %}
{% block content %}
  <h1>Index</h1>
  <p class="important">
    Welcome to my awesome homepage.
  </p>
{% endblock %}
```

This can be useful for sharing a common page structure across a website.

## Step 6:

Finally build an HTTP form served on the route `"/films/submit"` that fulfils the function of step 4.

If you've got this working feel free to experiment with the include/excludes templating structures introduced today or play around with the features listed on the [Jinja Template Designer Documentation](#).

How did it go?

# Thank You!



Please complete the feedback survey by  
scanning the QR code or clicking [here](#)